Date __/__/____

Saathi

Recursion & Back Tracking

Any function which calls itself is called recursion. A recursion method solves a problem by calling a copy of itself to work on a smaller problem. Each time a function calls itself with a slightly simpler version of the original problem. This sequence of smaller problem must eventually converge on a base case.

working of recursion approach by summarizing the above three steps.

(1) Base Case / Terminating Condition: A recursion function must have a terminating condition at which the process will stop itself. Such a case is known as the base case. In the absence of a base case, it will calling itself and get stuck in an infinite loop. until the memory stack will full & stop the recursion.

(2) Recursive Call (Smaller problem): The recursive function will invoke itself on a smaller version of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is.

(3) Self-work: Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

Saathi

Date ___ / ___ / _____

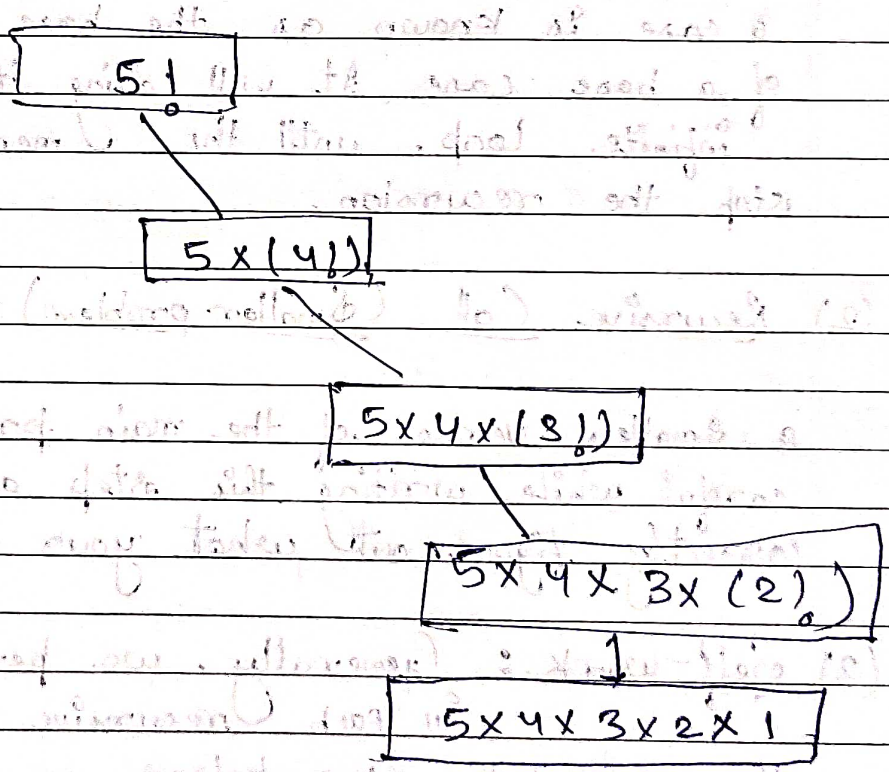
Note* → Recursion uses an in-built stack that store recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory overflow. If the number of recursion calls exceeded the maximum permissible amount, the recursion depth* will be exceeded. This condition is called stack overflow.

Now let us see how to solve a few common using Recursion.

1. Factorial. {5}

$n! = n*(n-1)*(n-2)*\cdots*1$

Fun(n) → if n==1 return 1;
return Fun(n-1)*n;



Date ___/___/_____

Saathi

Types of Recursion :- They are mainly of Two type

(1) Direct Recursion
- Linear
- Head.
- Tail / Non Tail.
- Tree.
- Binary
- Multiple Recursion

(2) Indirect Recursion
- → Mutual.
- Implicit Recursion
  - → Structured.
  - → Unstructured.

Direct Recursion → It occurs when a function calls itself directly.

Indirect Recursion → It occurs only when a function calls another function which eventually calls the first function.

Direct Recursion's are of 6 or more types.

(1) Linear Recursion → When it make single recursive call example (ascending / Descending).

```
fun(int n){
    if (n==0) return;
    print(n);
    fun(n-1);
}
```

(2) Binary Recursion → When a function makes two recursive calls.

example (fibonacci Sequence).

Page No. ____

Saathi

Date ___ / ___ / _____

```
int fun(int n){
    if(n<=1) return n;
    return fun(n-1)+fun(n-2);
}
```

'3' Multiple Recursion → When a function calls more than two function calls.

example {trifibonacci sequence}

```
int fun(int n){
    if(n==0) return 0;
    if(n==1||n==2) return 1;
    return f(n-1)+fun(n-2)+fun(n-3);
```

'4' Tree Recursion → When a function calls are multiple that they will form Tree like structure

```
int tree(int n){
    if(n<=1) return n;
    return tree(n-1)+tree(n-2);
```

'5' Head Recursion → When Recursive calls are made first rather than creating others statements except base condition

```
void headf(int n){
    if(n==0) return;
    headf(n-1);
    print(n);
}
```

Page No.

Date __/__/____

Saathi

'6' Tail & Non Tail Recursion → In tail recursion all call are made in the end of the function Tail recursion can be optimized by the compiler to avoid stack overflow.

```
void fun(int n){
    if(n==0) return;
    print(n);
    fun(n-1);
}
```

Non-Tail Recursion → It occurs when recursive call is not the last operation in the function, preventing tail call optimization.

```
int nonTail(int n){
    if(n==0) return 1;
    return n * nonTail(n-1);
}
```

* Indirect Recursion Types.

(1) Mutual Recursion → Mutual Recursion involves two or more functions calling each other in a cycle.

function prototype needed in this;

```
bool isEven(int);
bool isOdd(int);

bool isEven(int n){
    if(n==0) return true;
    return isOdd(n-1);
}
```

Page No.

Date __/__/____

Saathi

```
bool isOdd(int n) {
    if(n==0) return false;
    return isEven(n-1);
```

we will cover implicit in the end.

Back-Tracking

Backtracking is an algorithmic paradigm for solving problem incrementally, one piece at a time, and removing those solutions that fails to satisfy the constraints of the problem at any point. It is used in problems where a sequence of choices can lead to a solution or to a dead-end, and it explores all potential paths until it finds a valid solution or exhaust to all possibilities.

Key Concepts of backtracking.

1. Choice: At each step, choose an option from a set of possible optains.
2. Constraints: Check if the chosen optoin leads to a feasible solution.
3. Goal :- Determine if the current path meets the problem's goal.

Types of Backtracking

1. Simple Backtracking.
2. (CSP) Constraint Satisfaction problems Backtracking
3. Branch & Bound.

Page No.

Date __/__/____

Saathi

1. Simple Backtracking → It involves all possible solution and backtracking whenever a solutions fails to meet the constraints.

Example → N Queens problem.

2. CSP Backtracking → CSP backtracking is used to solve problems where a set of variables need to be assigned values that satisfy specific constraints.

Example → Sudoku Solver.

3. Branch & Bound →

Branch & Bound is used for optimization problems where we need to find the best solution accordingly to some criterion. It involves branching possible solutions & bounding them by estimating the lower & upper bands of the solution.

Example → Knapsack Problem.